Corrigé TP7

1. On lit la chaîne à partir de $i+1$ (on peut puisque par hypothèse, on sait déjà que le caractère i existe et est un chiffre) jusqu'à rencontrer soit la fin de la chaîne, soit un caractère qui n'est pas un chiffre. Puis on exploite String.sub pour extraire le nombre ainsi identifié.

   Rappel : String.sub s i l extrait de s la chaîne commençant à l'indice i et de longueur l.

```
let isoler_nombre (s:string) (i:int) :lexeme*int =
  let n = String.length s in
  let j = ref (i+1) in
  while (!j < n) && ('0' <= s.[!j]) && (s.[!j] <= '9') do
    j := !j+1
  done;
  CONST (int_of_string (String.sub s i (!j-i))), (!j-1)
```

2. On commence par exclure le cas où on se trouve en fin de chaîne : le lexème correspondant est EOF. On peut maintenant considérer le i-ème caractère de s et l'analyser selon les cas. Les deux cas non triviaux sont :

   - Le caractère en début de chaîne est un espace. On l'ignore en appelant récursivement premier_lexeme à partir de l'indice suivant dans la chaîne.
   - Le caractère en début de chaîne est un chiffre. Alors il faut extraire le nombre commençant cette position ce que permet précisément la fonction isoler_nombre.

```
let rec premier_lexeme (s:string) (i:int) :lexeme*int =
  let n = String.length s in
  if i = n then EOF, n
  else match s.[i] with
         |'(' -> LPAR, i
         |')' -> RPAR, i
         |'+' -> PLUS, i
         |'-' -> MINUS, i
         |'*' -> TIMES, i
         |'/' -> DIV, i
         |' ' -> premier_lexeme s (i+1)
         |x when ('0' <= x) && (x <= '9') -> isoler_nombre s i
         |_ -> raise Erreur_lexicale
```

3. On utilise premier_lexeme récursivement de la façon suivante dans analyse_lexicale :

   - Si le premier lexème identifié n'est pas EOF, on le stocke dans un accumulateur puis on reprend récursivement l'analyse à partir de la position suivant immédiatement la fin de ce premier lexème, valeur qui est justement fournie par premier_lexeme.
   - Sinon, c'est que l'analyse lexicale est terminée : on ajoute le dernier lexème EOF à l'accumulateur. Comme toute les adjonctions ont été faites récursivement en tête de liste, l'accumulateur contient à ce stade la liste des lexèmes associée à s en ordre inverse : il suffit de la renverser pour obtenir le résultat souhaité.

   On appelle analyse_lexicale à partir de la position 0 dans s et avec un accumulateur initialement vide. Les éventuelles erreurs lexicales sont correctement gérées grâce à premier_lexeme.

```
let lexeur (s:string) :lexeme list =
  let rec analyse_lexicale (acc:lexeme list) (position:int) :lexeme list =
    match premier_lexeme s position with
      |EOF, _ -> List.rev (EOF::acc)
      |lexeme, fin_lexeme -> analyse_lexicale (lexeme::acc) (fin_lexeme+1)
  in analyse_lexicale [] 0
```

4. On explique le fonctionnement de parser_E ; celui des deux autres fonctions étant similaire. Le premier lexème d'un mot dérivant de E ne peut être que CONST n, MINUS ou LPAR d'après les règles de la grammaire. Si la liste ℓ commence par autre chose on peut donc lever l'exception Erreur_syntaxique. On a maintenant trois cas à traiter :

   - Si la tête de ℓ est CONST n, ce dernier est déjà un mot engendré par G : on renvoit donc son arbre syntaxique et la queue de ℓ.
   - Si la tête de ℓ est MINUS alors il faut réussir à identifier quelle est l'expression arithmétique dont on est en train de considérer l'opposé. Pour ce faire, on appelle récursivement parser_E puisque les règles indiquent que MINUS ne peut être suivi que d'un mot dérivant de E.
   - Si la tête de ℓ est LPAR, alors les règles indiquent que le mot qui suit doit dériver de B et qu'on doit avoir juste après le lexème RPAR. On appelle donc parser_B sur la queue de la liste : ceci renvoit, sauf en cas d'erreur syntaxique, un arbre syntaxique et une nouvelle liste ℓ' dont on vérifie que la tête est bien égale à RPAR. Si ce n'est pas le cas, on déroge à la règle $E \to$ LPAR B RPAR et on peut lever Erreur_syntaxique.

```
let rec parser_E (l:lexeme list) :ea*lexeme list =
  match l with
  |(CONST n)::q -> Const n, q
  |MINUS::q -> let e,r = parser_E q in Oppose e,r
  |LPAR::q -> (match parser_B q with
               |e, RPAR::r -> e,r
               |_ -> raise Erreur_syntaxique)
  |_ -> raise Erreur_syntaxique
  and parser_B (l:lexeme list) :ea*lexeme list =
    let e1,r1 = parser_E l in
    let op,r = parser_O r1 in
    let e2,r2 = parser_E r
    in Bin(op,e1,e2), r2
  and parser_O (l:lexeme list) :op_binaire*lexeme list =
    match l with
      |PLUS::q -> Plus,q
      |MINUS::q -> Moins,q
      |TIMES::q -> Fois,q
      |DIV::q -> Div,q
      |_ -> raise Erreur_syntaxique
```

   Remarquez que seules parser_E et parser_B s'appellent récursivement l'une l'autre : on aurait pu écrire parser_O à part.

5. Un mot engendré par G l'est nécessairement via une dérivation commençant par $S \Rightarrow E$ EOF : on appelle donc parser_E sur l'entrée : si le résultat de cet appel indique que le seul lexème restant à analyser est EOF, l'analyse syntaxique termine sans erreur, sinon, on lève Erreur_syntaxique.

```
let rec parseur (l:lexeme list) :ea =
  match parser_E l with
    |e, [EOF] -> e
    |_ -> raise Erreur_syntaxique
```

6. On commence par écrire une fonction evalue_ea premettant d'évaluer une expression arithmétique à partir de son arbre syntaxique. Très classique.

```
let rec evalue_ea (e:ea) :int =
  match e with
  |Const n -> n
  |Oppose e -> - (evalue_ea e)
  |Bin(Plus,e1,e2) -> (evalue_ea e1) + (evalue_ea e2)
  |Bin(Fois,e1,e2) -> (evalue_ea e1) * (evalue_ea e2)
  |Bin(Moins,e1,e2) -> (evalue_ea e1) - (evalue_ea e2)
  |Bin(Div,e1,e2) -> (evalue_ea e1) / (evalue_ea e2)
```

   Puis, on utilise en cascade lexeur, parseur et evalue_ea dans l'ordre indiqué dans le plan de bataille.

```
let rec evalue (s:string) :int = evalue_ea (parseur (lexeur s))
```

   Remarque : A la place de fournir les chaînes d'expressions arithmétiques directement on pourrait aussi aller les lire dans un fichier et les évaluer grâce à evalue.

7. Je vous laisse vous entraîner ! Pour vous aider, vous pouvez reprendre la structure de ce TP : définition des types dont on aura besoin, découpage en analyse lexicale et analyse syntaxique, identification de la grammaire à considérer...